Processes and Structured Transition Rules for Case-Based Reasoning

Peter Funk

Mälardalen University, Department of Computer Science and Engineering
P.O. Box 883, Västerås, Sweden
peter.funk@mdh.se

Abstract

Traditional notations of processes are often based on notations that are formalized with different types of automata and a suitable domain specific (graphical) notation. These notations allow elaborate analysis of the processes such as proving equivalence between processes. They are also efficient in capturing exact behavior, but are sometimes regarded as difficult to use by users less familiar with formal notations and do not always promote easy reuse and stepwise refinement.

A notation for processes using transition rules and categorization of formulae in transition rules is exemplified and proposed for applications domains such as business processes, system development processes etc. Transition rules enable users to successively refine the processes and also enable similarity measurements and reuse in a Case-Based Reasoning approach. Transition rules in the proposal handle also time, important for some features. Using formalized transition rules also gives access to logical based tools (verification, simulations).

Introduction

Processes are increasingly popular in company's and are used to store experience. These processes have often been refined and improved during long use and capture essential experience gained by employees (failure and successes). Such processes may capture how to handle unsatisfied customers, resolve conflicts with business partners, reorganize the organization or business modelling (Chen-Burger and Robertson 1998) and (Watson I. 1997). These processes reflect a companies experience in a variety of issues and may be a key to a company's success or failure. Processes may capture experience in a wide area of applications, e.g. product development, customer handling, business processes, employment processes and maintenance instructions. Individual work instructions and work flow diagrams may be stored together with the tasks. Task are building block of processes (explained further on).

The captured skill and experience is expensive to gain and is often acquired over a long period of time and building experience bases is an important issue in most areas, e.g. in software engineering. For more on CBR in software engineering, see Althoff, Birk, Wangenheim, Tauz 1998. Half of all software development projects fail (Sommerville 1994) so less successful cases are a large part of the experience and should also be kept in the case library to avoid repetition of past failures. Development processes are often used to identify quality and lead-time problems at an early stage by providing clear checkpoints for project members, project leaders and management. Many large companies in technical intense areas have tens of thousands of people working in different development projects. Savings are considerable if experience can be transferred efficiently between these people using techniques such as Case-Based Reasoning.

A generic internal notation for processes that enables tailoring, validation, verification and reuse of processes is proposed. This notation has been successfully applied in a CBR system developed for the domain of telecommunication services, called CABS, Case-Based Specification (Funk 1998). The notation confronting the user should be adapted to the application domain and should follow users requests and state of the art (graphical) notations for the domain (this is not further explored in this paper). When reuse of a task occurs, previously collected experience that is captured in the reused parts originating from e.g. local improvements, successful and less successful experience, extension with metrics and standards is also retained. If the user changes a process or creates a new process for some purpose, this is stored in the case library together with feedback on how well the outcome of the modified or new process was. This paper is part of the work to extending the CABS system to new application domains.

Formulae

Before defining processes, tasks and cases we define formulae. The next section describes tasks and transition rules where formulae are used (condition and conclusion part). Formulae are attributes and relations as used in predicate logic, they may be true in a given moment or false. Example of a conjunction of predicates (non decomposable formulae may be negated with "no" and attributes with capital letters are instantiated or uninstantiated variables, for semantic see Funk 1998):

```
egg(6) & flour(2,dl) & milk(3,dl) &
butter(2,spoons) & frying_pan(1,teflon)
& gas_stove(g1) & bowl(b1) & bowl(b2) &
no gas(X) & no pancakes(X)
```

Every formula has a well defined meaning in the application domain. An ontology for the application domain will give the necessary means for a selecting formulae. In most domains the formulae may be divided in different categories. From a logical point of view there may not be a need for categorization of these formulae (predicates), but for some application domains categorization gives advantages in a case-based reasoning approach (different categories may have different significance in matching, se matching section). In the domain of telecommunication services (the first application domain for the CABS system) three categories where used, in-signals, out-signals and facts. In-signals are events and out-signals are not observable by the system and facts are true or false, and continue to be so until otherwise concluded in a conclusion part of an executed transition rule (see next section).

Tasks

Atomic tasks are transition rules and are the smallest units the case-based reasoning system handles and tasks may be reused individually or in sets (part of a process or a complete process description). The application domains targeted have time as an important feature, hence tasks have a strict time handling. If *t* is the current time and *dt* is the time needed to produce the conclusions, then *t+dt* is the time when the conclusions are available. "p" stands for provable. Conditions and conclusions (called input and output in the paper) are conjunctions of atomic formulae.

$$p(Conditions, t) \rightarrow p(Conclusions, t+dt)$$

Atomic tasks do not contain other tasks (further on atomic tasks are called "task") and are the finest granularity of causal relations used for describing processes. In our example in Table 1 an example on how atomic formulae in condition and conclusion part are divided into categories. The ontology for the application domain should determine these categories (dynamic assignment of categories is not implemented in the prototype). The example structure in Table 1 is: 1) `Name` (a unique name naming the task). 2) `Resources` is a conjunction of items that are consumed, food items are consumed during dinner preparation. The consumption of resources may be a domain rule. 3) `Tools` is a conjunction of items that are considered more permanent in the domain, e.g. `frying pan & stove & bowl & chef.` Tools are by default available after a task is completed (application domain rule). There may be occasions where a task removes or creates a tool (worn out, replaced, built etc.). Relations are formulae with two or more attributes are used to describe relations, e.g. `replaces(soya_flour, egg)`.

If the conditions in the task are provable at time t, output is necessarily provable at time t+dt. The time dt is the time it takes to complete a task is individually calculated or estimated (based on previous experience) for each task.

```
Task: prepare pancake batter
Task content (plain text): mix ingredi-
   ents in bowl until batter is smooth.
Resource: 1 egg & 2 dl flour & 3 dl
   milk, 2 spoons butter
Tools: bowl & mixer & spoon & person
->
Output: pancake batter & dirty bowl &
   dirty mixer
Time to prepare: 2 minutes
```

Table 1. An example of a task (syntax eased)

In the example person (e.g. a chef) may be considered as an odd tool, and it may be worth to consider if a category `agents` is appropriate. Agents being able to interact with their environment, with different level of abilities and experience, e.g. an experienced chef would be able to creatively solve different problems perhaps replacing eggs with Soya flour if there is lack of eggs or a vegetarian amongst the guests (if this is not known as a application domain fact). However tools and agents are similar in our domain in that they both still are available after a task has been completed (the chef and frying pan are both available for other tasks once the task has been completed). The *task content* is dependent on the domain (and the actors), if it is a robot then the task content may be a program, in industry it may be a work flow diagram, for a amateur chef it may be plain English with illustrations.

For more on the formal notation of logic used see (Funk 1998). There is no need for the user to know the formal notation. The semantic link to the application domain is maintained with application domain specific notations and terminologies (graphical/textual). The connection to domain specific notations is not further explored in this paper.

In Figure 1 an example of a graphical notation for a task is shown (used in the graphical process example in Figure 2). *Task definition:* the transition rule with conditions and conclusions. *Task description:* is an informal description of the task. *Task information:* may be information such as record on how successful the application of the task has been and even comments, suggestions from previous users.

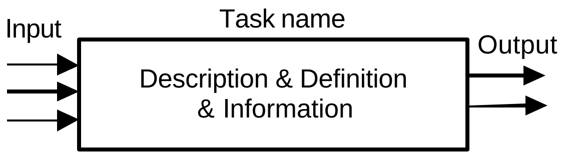

Figure 1. Task with input, name and output

Composed Tasks

In many domains it is often natural to group a number of tasks in one "unit". A composed task is such a unit of tasks (composed or atomic tasks). In this way a process may be described in layers. Layering is often a user request and used in successful state based approaches, e.g. state charts. This enables a recursively layered structure of tasks (no self reference is allowed, which can be automatically detected since formalism is based on predicate logic). The current prototype handles two layers and will be extended to handle any level of layering. It is always possible to expand all decomposable tasks until a process only contains atomic tasks. There may be different ways to solve a problem since the same result may be achieved in different ways using different tasks.

Process Notation and Cases

Processes are represented as a set of tasks that are ordered in such a way that the process produces the output given the available input (resources, tools, etc), costs constraints and time constraints (Figure 2). Input to a process is the sum of all input required by the tasks after reduction of internally produced input (a,b,c,d).

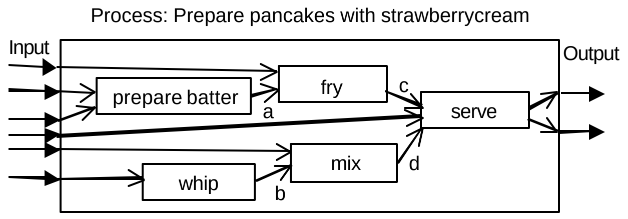

Figure 2. Example of a process (simplified)

Every previously applied process and all tasks in the process will be stored as a case (size of case library is not an issue for the application domains considered). A process can be seen as decomposable task. It is more likely that parts of cases will be reused than full cases. The reason is that a new problem having similar or even the same output to a previous case most likely has different circumstances, restrictions and available resources, hence parts need to be modified or replaced.

Cases may also store additional information on where, when and how many times the case has been reused and how successful the case was etc.

Problem Description

The problem description may be a more or less completed task or process descriptions extracted from the users problem description and a given problem context (what resources and tools are currently available) and domain (domain specific rules). In some application domains only the output may be given and the solution is a process able to produce the output together with a list of resources and tools needed to produce the output. In other application domains some desired properties for tasks or subprocesses may be given (e.g. use only ISO9000 certified tasks for welding pipes).

Matching and Reuse

An outline of retrieval and reuse is given (see Figure 3).

Step A: In step *A* the input problem is translated (or used to induce) a set of *skeleton tasks*. A *skeleton task* is an incomplete task in respect to missing or even incorrect conditions and conclusions. A task stored in the case library is expected to include all input and output, be validated, verified and previously used in a case. Application domain knowledge may be used to fill in or modify parts of the skeleton tasks or give the user suggestions to improve the problem description.

Step *B*: In step *B* the skeleton tasks are matched with cases in the case library. This is a set based matching, and for each formulae category in the condition and conclusion part, a triple value is assigned (intersection, disjunction between the skeleton task category and transition task category and transition task category and skeleton task category). These triplets give a set of values describing similarity between tasks. If in the particular application domain or specific problem resources are more important than tools, the triplet value for resources may be given a higher weight (a application domain where resources are the bottleneck but tools can be fabricated, borrowed or bought if necessary). The triplets are summed for each matching tasks in the case library are ranked according to their sum, an approach that gives good results for some domains, shown in (Funk 1998). For each skeleton task a ranked list of matching tasks is produced.

Step *C*: These lists are given as input to step *C* that ranks the cases that best match the problem description and skeleton tasks. Each case is assigned a number according to how many skeleton tasks have a good scoring counterpart in the matching case (process). A threshold value for "good" matching tasks must be set by the user (the performance of the matching has in the domain of telecom services shown to be very insensitive to a wide range of threshold values).

Step *D*: In step *D* the best case is selected or if there are more cases covering different parts of the input problem, appropriate parts may be merged. If there are still parts from the problem description that are not covered, tasks may be identified to fill in the missing parts. Induction is another possibility and new tasks can be constructed, filling in the missing bits (may require human interaction).

In all three steps we use *domain* knowledge and *problem context* (illustrated by containers in Figure 3). One example for *B* and *C* is that if a matching case requires tools that are not available (tools may have a cost associated with them) it is scored lower than a task with similar input and output, requiring only tools that are available but missing some resources. Domain rules capture information for the domain, e.g. which resources are replaceable with other resources (the egg and Soya flour example).

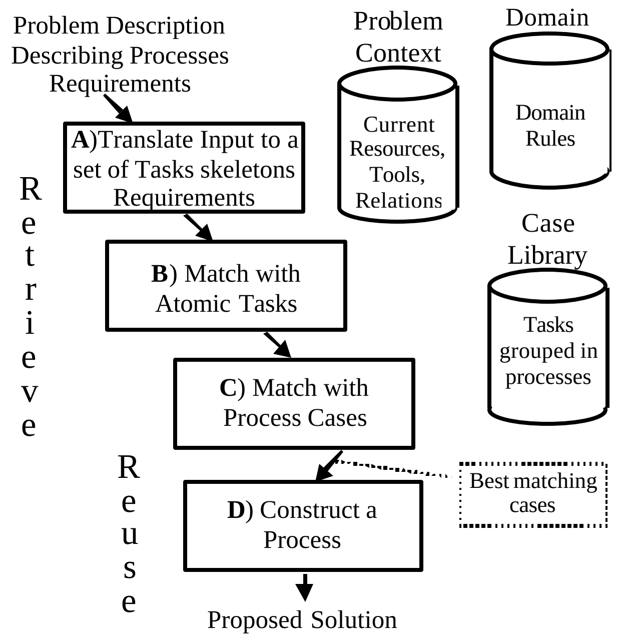
Problem Description
Describing Processes
Requirements
Problem Context
Domain
A)Translate Input to a set of Tasks skeletons Requirements
Current Resources, Tools, Relations
Domain Rules
Retrieve
Case Library
B) Match with Atomic Tasks
Tasks grouped in processes
C) Match with Process Cases
Reuse
Best matching cases
D) Construct a Process
Proposed Solution

Figure 3. Outline of retrieval and reuse

Revision and Restore

Since the tasks are formalized and the problem context and domain are available, the proposed solution can be simulated by the user. These simulations may be stored with the solution case (such test-cases may be essential in a CBR approach where cases may be modified or evolve, e.g. being software functions, see Minor and Hanft 1999). Storing cases in the case library is made in a traditional way, mainly with some generalization performed on tasks. How successful a solution is should also be stored with the new case and its history (from where it stems, modification records, how often reuse has occurred, etc.).

Conclusions

This paper proposes that formulae in transition rules are categories to improve matching. This modification will generalize the CABS system to handle other domains than telecommunication services (the categories *in-signal*, *out-signal* and *fact* used in the first version of CABS are highly domain specific). Enabling a structure of categories selected by the user and used in the matching process, to handle different significance for different categories, is proposed. Decision on what categories are used should be based on the ontology for the application domain. Testing the approach on new application domains will be the main focus on future work in CABS.

Tasks are also proposed to be layered in more than two layers. These extensions have not been fully implemented in the prototype but it is argued that they are reasonable. Categories improved the matching in CABS for telecommunications services. An example of different significance for categories is if resources is the main constraint in the domain, but tools can be borrowed if missing (e.g. cooking food in a student dormitory, ingredients are more precious since student are short of money, but kitchen tools may easily be borrowed from some other student). The matching process should be directed towards solutions where resources differ less from what is available and differences in tools should influence the ranking of the proposed solutions less.

The matching is carried out in two steps, first tasks are matched and thereafter processes. The formalization in tasks also enables consistency checks, verification and validation by simulation of the proposed solution.

References

Althoff K.-D., Birk A., von Wangenheim C. G. and Tautz C. 1998. CBR for Experimental Software Engineering. In *Case-Based Reasoning Technology: From Foundations to Applications*. Lenz M., Bartsch-Spörl B., Burkhard H.-D., Wess S. (eds). Springer.

Chen-Burger Y.-H., Robertson D. 1998. Formal Support for an Informal Business Modelling Method. In *Proceedings Tenth International Conference on Software Engineering and Knowledge Engineering, SEKE'98*.

Funk, P. 1998. CABS: A Case-Based and Graphical Requirements Capture, Formalisation and Verification System. Ph.D. diss, Department of Artificial Intelligence, University of Edinburgh.

Funk, P. and Crnkovic, I. 1999 Case-Based Reasoning for Reuse and Validation of System Development Processes. In *Challenges for Case-Based Reasoning, Proceedings of the ICCBR'99 Workshops*. Schmitt S.,Vollrath. I., (eds). University of Kaiserslautern.

Minor M. and Hanft A. 1999. Cases with a Life-Cycle. In *Challenges for Case-Based Reasoning, Proceedings of the ICCBR'99 Workshops*. Schmitt S.,Vollrath. I., (eds). University of Kaiserslautern.

Sommerville I. 1996. *Software Engineering*. Fifth edition part 1, Addison Wesley.

Watson I. 1997. *Applying Case-Based Reasoning: Techniques for Enterprise Systems*, Morgan Kaufmann.